О подходе к реализации универсального интерпретатора на основе L-преобразователей

А.А. Николаев

***Аннотация* – Рассматривается разработка универсального интерпретатора, который настраивается на конкретный язык программирования с помощью специального графового описания. Лексический и синтаксический уровни языка представляются наборами L-графов, а правила перевода программы во внутреннее представление задаются L-преобразователем с действиями. В качестве промежуточного представления используется постфиксная последовательность команд, исполняемая стековой виртуальной машиной. Описаны общая схема обработки программы, формат спецификации языка и архитектура программного прототипа. Реализация включает графический редактор, загрузку и проверку описания языка, лексический и синтаксический анализ, формирование программы на языке стековой машины и ее выполнение. Работа проиллюстрирована на примере, в котором программа состоит из одного числа, а результатом является это число, увеличенное на единицу. Рассматриваются поддерживаемые конструкции входных языков и направления дальнейшего развития системы.**

***Ключевые слова* – формальный язык, L-граф, L-преобразователь, универсальный интерпретатор, стековая машина, синтаксический анализ**

I. Введение

При разработке нового языка программирования обычно приходится решать несколько связанных задач: описывать лексику, синтаксис и семантику, а также реализовывать исполнение программы. В традиционной схеме реализация этих этапов нацелена на один конкретный язык. Даже небольшое изменение синтаксиса может потребовать правок в лексическом или синтаксическом анализаторах и последующих этапах обработки.

Программные инструменты автоматической генерации синтаксических анализаторов частично снимают эту проблему. Средства семейства генераторов Yacc и более современные системы, такие как ANTLR, позволяют задавать грамматику в текстовой форме и получать готовый анализатор [1], [2]. Однако после синтаксического анализа разработчику по-прежнему необходимо определить внутреннее представление программы, правила его построения и механизм исполнения. Такие инструменты удобны при создании компиляторов, но сами по себе не являются готовой средой, в которой можно задать язык и сразу выполнить программу на этом языке. Они решают прежде всего задачу построения анализатора, но не задают автоматически модель памяти, правила трансляции, механизм выполнения функций, ввод-вывод и другие элементы интерпретации программы.

В зарубежной литературе близкие к рассматриваемой нами идее универсального интерпретатора развиваются в нескольких направлениях: generic interpreters (обобщенные, или параметризуемые, интерпретаторы), executable semantic frameworks (исполняемые семантические фреймворки, то есть программные каркасы для задания семантики языка в исполняемой форме), language workbenches (инструментальные среды для разработки языков) и definitional interpreters (интерпретаторы-определения, с помощью которых задается семантика языка). Например, в системе nitrO [3] описан обобщённый интерпретатор, не зависящий от конкретного языка, который получает программу и спецификацию языка. В [4] рассматривается универсальный интерпретатор, параметризуемый формальным описанием семантики языка. Система K Semantic Framework [5] и PLT Redex [6] позволяют задавать исполняемые формальные семантики языков программирования, а Spoofax [7] рассматривает декларативное описание языков и связанных с ними инструментов разработки. Исторически к этому направлению примыкают definitional interpreters (определяющие интерпретаторы, то есть интерпретаторы, задающие семантику языка), где интерпретатор используется как способ задания семантики языка [8]. Отдельно можно отметить Truffle [9], предоставляющий общую виртуальную машину и оптимизирующую инфраструктуру для языков, реализованных через AST-интерпретаторы. Описание нового языка в этой системе обычно не является внешней графовой спецификацией: для каждого языка разработчик реализует собственный набор узлов AST, правила разбора и семантику на языке реализации. Поэтому Truffle ближе к общей платформе выполнения и оптимизации многих языков, чем к интерпретатору, который напрямую читает одну внешнюю спецификацию языка и исполняет программу по этой спецификации.

В настоящей статье предлагается другой вариант подхода к реализации универсального инструмента для интерпретации с помощью L-графов. Принципиальное отличие состоит в том, что описание языка задается не только текстовой грамматикой и не программным кодом отдельного AST-интерпретатора, а графовой спецификацией. Лексический и синтаксический уровни представляются наборами именованных L-графов, а правила перехода от разобранной программы к исполняемому внутреннему представлению задаются L-преобразователем с действиями. Один и тот же программный конвейер читает эту спецификацию, выполняет анализ программы, строит постфиксное представление и передает его стековой машине. Таким образом, изменяемой частью является описание языка, а не ядро интерпретатора. Кроме того, действия преобразователя привязаны к дугам графа и задают не произвольные вставки кода, а контролируемые операции формирования ПОЛИЗ [10] и служебных структур транс-

Статья подготовлена по материалам магистерской диссертации.

Николаев Андрей Андреевич, факультет вычислительной математики и кибернетики МГУ имени М.В. Ломоносова. E-mail: Nikas_0604@mail.ru.

ляции. L-графы развивают идею графового представления конечных автоматов и позволяют описывать более сложные зависимости за счет скобочных пометок на дугах [11], [12], [13].

Цель работы состоит в проектировании и реализации программной системы, которая получает описание языка, исходную программу и входные данные, после чего выполняет лексический и синтаксический анализ, формирует постфиксное внутреннее представление и запускает его на стековой виртуальной машине. Пользователь при этом работает не с заранее зафиксированным языком, а с инструментом для его задания.

Основные результаты работы следующие. Предложена общая архитектура универсального интерпретатора на основе L-графов и L-преобразователей; разработан формат спецификации, объединяющий машинное и графическое описание языка; реализован программный прототип с веб-интерфейсом; подготовлены примеры языков и программы для проверки функций, рекурсии, циклов и ввода данных.

Структура статьи следующая. В разделе II приводятся необходимые сведения об L-графах и L-преобразователях. В разделе III описывается модель универсального интерпретатора. В разделе IV рассматривается программная реализация и поддерживаемые конструкции входного языка. В разделе V приводятся примеры работы и результаты функциональной проверки. В разделе VI обсуждаются ограничения и направления развития подхода.

II. L-графы и L-преобразователи

A. *L-графы*

Согласно [11] L-графом будем называть шестерку

$$G = \langle V, \Sigma, P, E, I, F \rangle, \tag{1}$$

где V – конечное множество вершин, Σ – алфавит основных символов, P – скобочное множество, E – множество дуг, $I \subseteq V$ – множество начальных вершин, а $F \subseteq V$ – множество заключительных вершин.

Дуга соединяет две вершины и может содержать основную пометку из $\Sigma \cup \{\varepsilon\}$, а также одну или несколько скобочных пометок. Поэтому L-граф внешне похож на диаграмму конечного автомата, но успешность пути определяется не только его начальной и заключительной вершинами: скобочный след пути должен образовывать правильную скобочную последовательность. Язык $L(G)$ состоит из основных пометок всех успешных путей графа.

Для задач обработки программ удобно рассматривать бесконтекстный подкласс L-графов. На лексическом уровне такие графы могут задавать числа, идентификаторы, строковые литералы и служебные слова. На синтаксическом уровне они описывают выражения, операторы, блоки, функции и рекурсивные конструкции. При этом вложенность отражается скобочными пометками, а не явными операциями со стеком, как в магазинном автомате [11], [12].

В практической системе объемный язык задается не одним графом, а набором именованных графов. Один граф может ссылаться на другой через пометку дуги. Например, граф `<число>` использует граф `<цифра>`, а граф `<программа>` – граф `<число>`. Такое разбиение делает описание языка модульным и позволяет переиспользовать уже заданные конструкции [14].

B. *L-преобразователь*

В работах [15], [16], [17] L-преобразователи рассматриваются как графовая форма задания алгоритмов, близкая по назначению к классическим формализациям вычислений, но более удобная для визуального описания и дальнейшей генерации программного кода. Для целей данной статьи будем пользоваться следующим рабочим определением.

Пусть задан входной алфавит Σ, выходной алфавит Ω и скобочное множество P. L-преобразователем будем называть восьмерку

$$T = \langle V, \Sigma, \Omega, P, E, I, F, \lambda \rangle, \tag{2}$$

где $\langle V, \Sigma, P, E, I, F \rangle$ является L-графом, а функция

$$\lambda : E \to \Omega^* \tag{3}$$

сопоставляет каждой дуге выходную цепочку. Если путь

$$\pi = e_1 e_2 \dots e_m \tag{4}$$

является успешным, то его входной пометкой считается конкатенация основных пометок дуг, а выходной пометкой – цепочка

$$\lambda(\pi) = \lambda(e_1)\lambda(e_2)\dots\lambda(e_m). \tag{5}$$

Тем самым L-преобразователь задает соответствие между входными цепочками и выходными цепочками. Если для одной входной цепочки существует несколько успешных путей, то в общем виде получается отношение, а не обязательно однозначная функция. В реализованном интерпретаторе используется выбранный успешный путь, соответствующий разбору программы по заданной спецификации.

C. *L-преобразователь с действиями*

Для построения интерпретатора одной выходной цепочки недостаточно. При переводе программы нужно запоминать имена переменных, строить метки переходов, обрабатывать параметры функций и формировать команды виртуальной машины. Поэтому в работе используется L-преобразователь с действиями.

Формально такой преобразователь можно задать как

$$T_a = \langle V, \Sigma, P, E, I, F, A, \alpha, \rho \rangle, \tag{6}$$

где $\langle V, \Sigma, P, E, I, F \rangle$ является L-графом, A – конечное множество допустимых действий, $\alpha : E \to A^*$ сопоставляет дуге последовательность действий, а ρ задает смысл каждого действия как операции над состоянием трансляции. Состояние трансляции включает уже построенную постфиксную запись, таблицу временных имен, стек служебных меток, информацию о текущей функции и другие вспомогательные данные.

Если обозначить состояние трансляции через s, а накопленный выход через $y \in \Omega^*$, то действие $a \in A$ можно рассматривать как частичную функцию

$$\rho(a) : S \times \Omega^* \to S \times \Omega^*. \tag{7}$$

Во время прохождения успешного пути действия выполняются в порядке обхода дуг. Поэтому преобразователь одновременно проверяет структуру программы и формирует ее внутреннее представление.

В данной реализации основная цель действий – перевод в постфиксную запись. Например, для выражения $x + 1$ действия сначала добавляют в выход переменную x, затем константу 1, затем операцию $+$. Получается последовательность для исполнения на стековой машине:

$$x \quad 1 \quad +. \tag{8}$$

Для управляющих конструкций действия дополнительно создают метки и команды условных или безусловных переходов.

III. Модель универсального интерпретатора

Универсальный интерпретатор получает на вход следующие данные:

1) набор L-графов, описывающих лексический уровень языка;
2) набор L-графов, описывающих синтаксический уровень;
3) L-преобразователь с действиями трансляции;
4) текст программы;
5) входные данные программы, если они необходимы.

Результатом работы является значение, выведенное программой, либо диагностическое сообщение. Ошибка может возникнуть при чтении спецификации, лексическом или синтаксическом анализе, формировании промежуточного представления и непосредственно во время исполнения.

Общая архитектура показана на рис. 1. Слева представлена клиентская часть: редактор L-графов, загрузка и сохранение файлов, поле исходной программы, окно результата и диагностические сообщения. Пользователь задает описание языка и текст программы, а дальше система выполняет внутренние этапы обработки.

Центральная часть отвечает за перевод во внутреннее представление. Сначала JSON-описание языка разбирается и превращается в модель языка. Затем проверяется корректность графов и запускается лексический анализ исходного текста. После этого L-преобразователь обрабатывает поток лексем и формирует семантические действия, на основе которых строится ПОЛИЗ. Дополнительно ведется таблица символов и метаданных, необходимая для переменных, функций и служебных меток.

Правая часть схемы – интерпретатор стековой машины. Он загружает ПОЛИЗ, выполняет команды, работает со стеком и встроенными операциями, а затем формирует итоговый результат. Такой порядок важен: описание языка и программа проходят путь от графов и правил к ПОЛИЗ, а затем к результату выполнения.

A. Этапы обработки программы

Сначала система загружает спецификацию и проверяет наличие обязательных разделов. Затем лексический анализатор преобразует исходный текст в последовательность токенов. Для каждого токена сохраняются тип, значение, исходная лексема, номер строки и позиция в строке. Эти данные используются как при разборе, так и при формировании понятных сообщений об ошибках.

Следующим этапом является синтаксический анализ. Анализатор сопоставляет поток токенов со стартовым правилом языка и строит промежуточную структуру программы. В формальной модели этот этап соответствует поиску успешного пути по синтаксическим L-графам.

После успешного разбора программа переводится во внутреннее представление, удобное для интерпретации на стековой машине. Такое уточнение позволяет явно представить загрузку констант, работу с переменными, вызовы функций и переходы.

Последний этап – запуск виртуальной машины. Она последовательно читает команды, изменяет стек и состояние текущего кадра функции, выполняет переходы и формирует вывод.

IV. Программная реализация

A. Представление спецификации

Внешнее описание языка хранится в формате JSON. Такой формат легко читать из программы, редактировать вручную и передавать между клиентской и серверной частями. Основные поля спецификации показаны в листинге 1.

Листинг 1. Сокращенная структура спецификации языка

```
{
 "meta": {"name": "IncrementOneLang", "version": "1.0"},
 "lexer": {
  "tokens": [{"name": "NUMBER", "pattern": "\\d+"}],
  "skip": [{"pattern": "[ \\t\\r\\n]+"}]
 },
 "entry": "Program",
 "rules": {"Program": {"type": "node"}},
 "graphs": {
  "lexical": [],
  "syntax": [],
  "transducer": {}
 }
}
```

Поле `lexer` содержит определения токенов, литералы, ключевые слова и правила пропуска пробельных символов. Поле `entry` задает стартовое правило. В разделе `rules` хранится нормализованное представление синтаксиса, а раздел `graphs` содержит графы лексического и синтаксического уровней и один L-преобразователь.

В текущем прототипе машинные правила и графическое представление хранятся рядом. Это решение было принято для поэтапной разработки: ядро интерпретатора можно проверять независимо от графического редактора, а графы использовать для проектирования и визуального контроля спецификации.

B. Поддерживаемый входной язык

Текущая версия универсального интерпретатора позволяет задавать языки, в которых есть основные конструкции небольшого алгоритмического языка [18].

На уровне выражений поддерживаются числовые, логические и строковые значения. Для них можно задавать арифметические операции, операции сравнения, логические операции и простые строковые действия. На уровне операторов поддерживаются присваивания, печать результата, условные операторы и циклы. Кроме того, в системе реализованы функции и рекурсивные вызовы, поэтому через спецификацию можно описывать не только линейные программы, но и программы с подпрограммами.

Практически это означает, что одна спецификация может задавать язык с привычными числами и операциями, другая – язык с другим синтаксисом блоков и функций, а

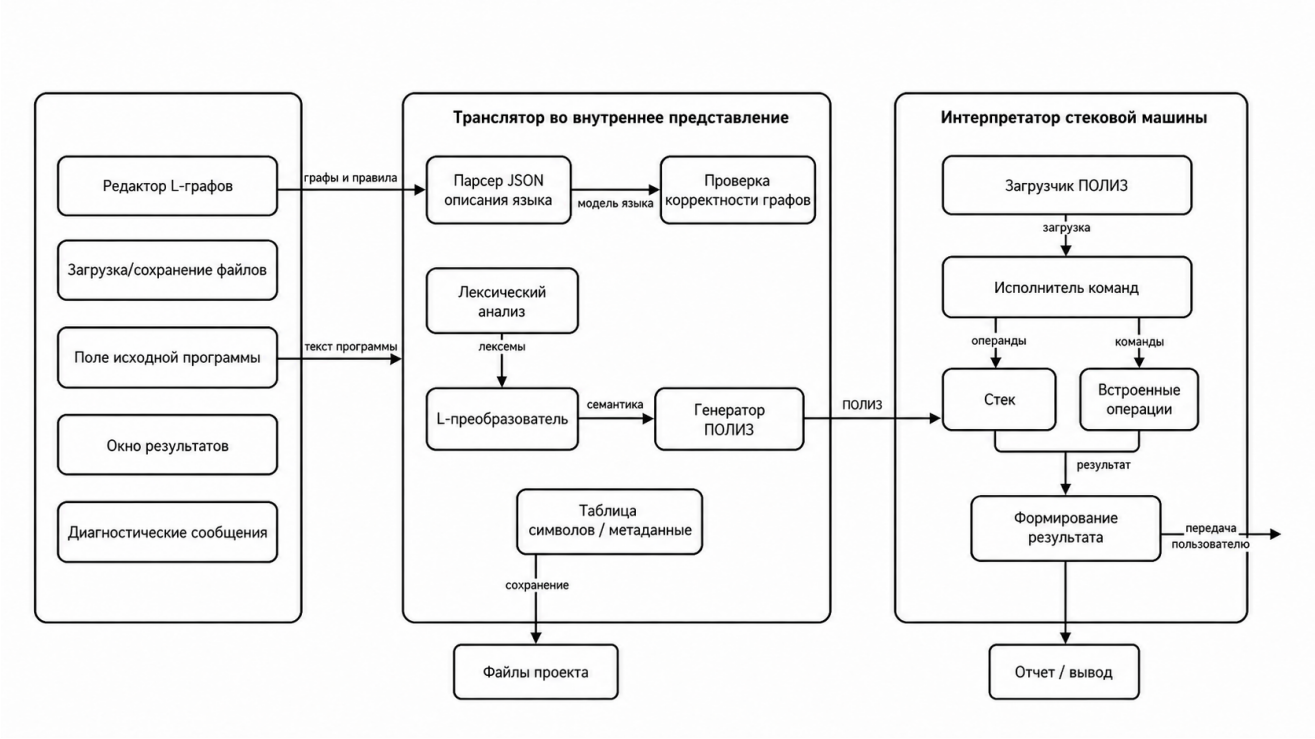

Рис. 1. Архитектура универсального интерпретатора

третья – язык, где числа записываются иначе. В демонстрационном языке палочной арифметики (унарной системы счисления) число представляется строкой из символов `1`, а операции сложения и вычитания сводятся к конкатенации и удалению символов.

Ограничения текущей версии связаны прежде всего с выбранной стековой моделью исполнения. В системе нет полноценной статической типизации, сложных составных объектов, развитой работы с динамической памятью и оптимизирующего этапа компиляции. Эти ограничения не мешают демонстрации основной идеи, но их нужно учитывать при переходе от учебных примеров к более громоздким языкам.

C. Действия L-преобразователя

Действия L-преобразователя можно разделить на две группы. Первая группа непосредственно добавляет элементы в постфиксную запись. Вторая группа не всегда порождает команду сразу, но управляет процессом трансляции: запоминает имена, собирает параметры, создает метки и переключает режимы обработки выражений.

Для простого языка инкремента (программе на вход подается число, а на выходе получается это число, увеличенное на единицу) достаточно такой последовательности действий:

```
OUT(<число>) -> OUT(1) -> OUT(+) -> OUT(PRINT).
```

Они формируют постфиксную последовательность

$$56 \quad 1 \quad + \quad \text{PRINT} \tag{9}$$

для входной программы `56`. В более сложных языках используется расширенный набор действий. Основные группы приведены в табл. I.

После формирования ПОЛИЗ он переводится в инструкции стековой машины. На этом уровне используются команды `PUSH_CONST`, `LOAD`, `STORE`, `BINARY`, `UNARY`, `CALL`, `RETURN`, `PRINT`, `JUMP` и `JUMP_IF_FALSE`. Такое разделение удобно: L-преобразователь работает с понятной постфиксной записью, а исполнительное ядро получает готовый набор инструкций.

D. Модули серверной части

Программный прототип реализован на Python 3 и состоит из нескольких модулей. Класс `GraphLexer` выполняет лексический анализ. Для каждой позиции выбирается самое длинное совпадение среди литералов и регулярных выражений. Если ни одно правило не подходит, анализ прекращается с указанием строки и столбца проблемного символа.

Класс `GraphParser` обрабатывает правила нескольких базовых типов: литерал, токен, ссылка на правило, последовательность, выбор, повторение, необязательный фрагмент и список с разделителем. Результаты разбора запоминаются по паре “имя правила – позиция во входе”, что уменьшает число повторных вычислений. Для построения узлов промежуточной структуры используются именованные захваты и преобразования.

Компилятор обходит полученную структуру и формирует инструкции стековой машины. Основные команды приведены в табл. II. Метки переходов сначала записываются символически, а после построения функции заменяются номерами инструкций.

Стековая виртуальная машина хранит отдельный кадр для каждого вызова функции. Кадр содержит локальные переменные, указатель текущей инструкции и стек вычислений. Для основной программы поддерживается таблица глобальных переменных. Встроенные функции обеспечивают ввод данных, вычисление длины и базовые преобразования типов. Также реализован вариант чисел в

Таблица I
Действия L-преобразователя, используемые при трансляции

Действие	Смысл в процессе перевода
`OUT(X)`, `emit(n)`, `emit(op)`	Добавить элемент в результирующий ПОЛИЗ: число, имя, операцию, команду печати или вызов.
`PUSH`, `LOAD`, `STORE`, `PRINT`, `CALL`, `RETURN`	Сформировать базовые команды стековой машины или элементы, которые затем разворачиваются в такие команды.
`save_var`, `save_callee`, `save_func`, `save_func_name`	Запомнить имя переменной, вызываемой функции или определяемой функции.
`save_params`, `append_param`, `ARG`, `ARG / CALL`	Обработать параметры функции и аргументы вызова.
`enter_expr`, `push_term`, `next_factor_group`, `push_op(*)`, `flush_ops()`	Управлять переводом выражений и порядком вывода операций в постфиксную форму.
`gen_lfalse`, `gen_lend`, `label_start`, `JZ`, `JMP`, `LABEL`	Создавать и использовать метки для условных операторов и циклов.
`cmp(num)`, `cmp(len)`	Настроить сравнение обычных чисел или сравнение по длине строки в палочной системе счисления.
`concat/remove`	Выполнить операции, специфичные для палочной арифметики: конкатенацию или удаление символов.
`dispatch`, `resume`, `end`	Управлять общим ходом трансляции и завершением обработки конструкции.

Таблица II
Основные команды виртуальной машины

Команда	Назначение
`PUSH_CONST`	поместить константу в стек
`LOAD, STORE`	прочитать или записать переменную
`UNARY, BINARY`	выполнить унарную или бинарную операцию
`JUMP`	перейти к заданной инструкции
`JUMP_IF_FALSE`	выполнить переход при ложном условии
`CALL, RETURN`	вызвать функцию и вернуть результат
`PRINT, POP`	вывести или удалить верхнее значение

виде строк из символов `1`; он используется в демонстрационном языке с палочной системой счисления.

Полный конвейер объединен в классе `Pipeline`. Он последовательно вызывает токенизацию, разбор, компиляцию и исполнение, а затем возвращает токены, промежуточную структуру, байткод и результат. Такое разделение полезно при отладке: любой этап можно запускать отдельно и сравнивать его выход с ожидаемым.

E. Пользовательский интерфейс

Серверная часть веб-приложения построена на FastAPI, клиентская – на HTML, CSS и JavaScript. Пользователь может редактировать JSON-спецификацию, исходную программу и входные данные в одном окне. Для графовой части предусмотрены три режима: набор лексических графов, набор синтаксических графов и L-преобразователь.

Редактор позволяет добавлять вершины и дуги, отмечать начальные и заключительные вершины, менять основную и скобочную пометки, задавать действие преобразования и перемещать элементы мышью. Для проверки предусмотрены отдельные вкладки с токенами, промежуточной структурой, байткодом и результатом исполнения.

V. Примеры работы

A. Язык одного числа

Для первого примера выбран язык `IncrementOneLang`. Корректная программа на этом языке состоит из одного целого неотрицательного числа. Результатом выполнения является значение программы, увеличенное на единицу. Пример намеренно сделан простым, чтобы на нем можно было проследить весь путь от графов до результата.

На лексическом уровне граф `<цифра>` позволяет распознать один из символов от 0 до 9. Граф `<число>` распознает последовательность цифр. Его начальная дуга переводит обработку в заключительную вершину, а петля позволяет прочитать оставшиеся символы числа. Синтаксический граф задает правило

$$\texttt{<программа>} ::= \texttt{<число>}. \tag{10}$$

L-преобразователь для этого языка показан на рис. 4. Первая дуга помещает входное число в выходную последовательность. Следующие дуги добавляют константу 1, операцию сложения и команду печати. После чтения конца файла путь завершается в заключительной вершине.

Для программы

```
56
```

концептуальная постфиксная запись имеет вид

$$56 \quad 1 \quad + \quad \text{PRINT}. \tag{11}$$

В программной реализации она разворачивается в следующий байткод:

```
0000: PUSH_CONST 56
0001: PUSH_CONST 1
0002: BINARY '+'
0003: STORE '__increment_result'
0004: PRINT
0005: LOAD '__increment_result'
0006: RETURN
```

После выполнения значение 56 и константа 1 оказываются в стеке. Команда `BINARY '+'` извлекает оба операнда и помещает обратно число 57. Это значение сохраняется, выводится пользователю и возвращается как результат основной функции.

B. Функциональная проверка

Помимо минимального примера были подготовлены языки с более сложным синтаксисом. `GraphFibLang` поддерживает блоки с ключевыми словами `begin` и `end`, определения функций, условные операторы, циклы и рекурсивные вызовы. `GraphStickLang` использует тот же управляющий синтаксис, но числа записываются последовательностями символов `1`. Язык `MiniL` демонстрирует, что одна и та же виртуальная машина может использоваться с другим набором ключевых слов, скобок и разделителей.

Результаты запуска подготовленных примеров приведены в табл. III.

Эти примеры не являются измерением производительности и не претендуют на полноту тестирования языка

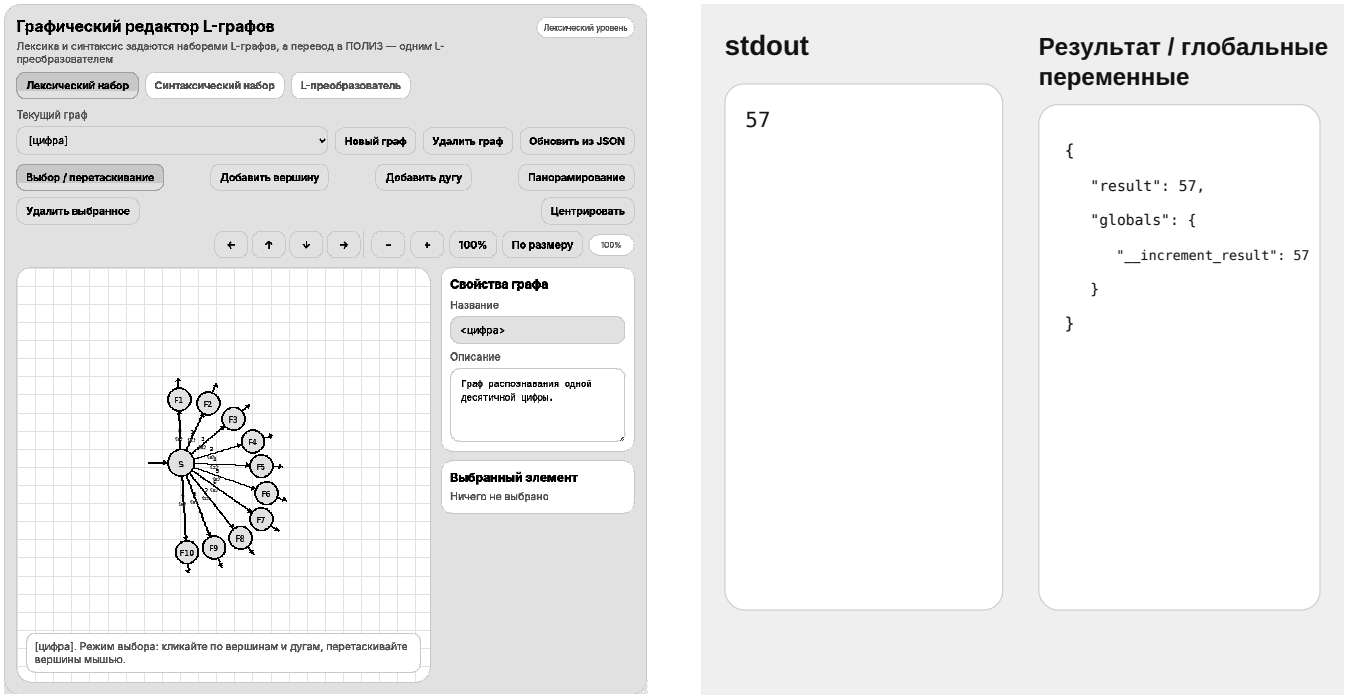

Рис. 2. Графический редактор L-графов и окно результата выполнения программы

(а) Лексический граф <число> (б) Синтаксический граф <программа>

Рис. 3. Графы демонстрационного языка

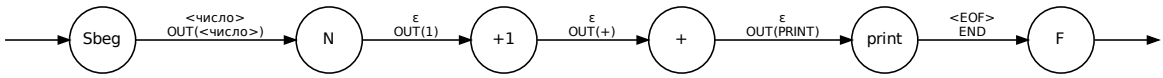

Рис. 4. L-преобразователь, формирующий постфиксную запись для языка `IncrementOneLang`

общего назначения. Их задача – проверить, что общий конвейер работает с разными спецификациями и поддерживает основные механизмы небольшого алгоритмического языка.

VI. Обсуждение и дальнейшее развитие

Главное преимущество предложенной архитектуры состоит в разделении языка и механизма исполнения. Лексика, синтаксис и правила трансляции находятся во внешней спецификации, тогда как стековая машина не зависит от конкретных ключевых слов и формы операторов. Благодаря этому можно экспериментировать с синтаксисом, не переписывая исполнительное ядро.

Графовое представление полезно не только как способ хранения правил, но и как средство анализа. На схеме видно, где начинается и заканчивается конструкция, какие альтернативы существуют и на каких переходах выполняются действия. Для небольших языков и учебных примеров это заметно упрощает обсуждение и отладку по сравнению с длинным текстовым описанием грамматики.

Предлагаемый интерпретатор не следует рассматривать как замену промышленным генераторам парсеров. Yacc, Bison и ANTLR обладают развитыми алгоритмами анализа, средствами генерации кода и большой экосистемой. Основное отличие универсального интерпретатора заключается в следующем: графовое описание, трансляция во внутреннее представление и непосредственное исполнение объединяются в одной экспериментальной среде.

У текущей реализации есть несколько ограничений. Во-первых, графы и нормализованные правила пока хранятся как связанные, но отдельные части спецификации. Для полностью графового интерпретатора требуется надежное преобразование L-графов в исполняемые правила и обратная проверка соответствия. Во-вторых, стековая машина ориентирована на простые значения и не содержит развитой модели памяти. В-третьих, компилятор пока не выполняет оптимизацию полученного байткода.

Первое направление дальнейшего развития – возможность выбора виртуальной машины. Сейчас используется стековая модель, потому что она естественно связана с ПОЛИЗ и проста для реализации. В дальнейшем можно

Таблица III
Результаты запуска демонстрационных программ

Язык	Проверяемые конструкции	Вход	Результат
IncrementOneLang	число и простое арифметическое действие	программа `56`	`57`
GraphFibLang	рекурсивная функция и условный оператор	`fact(6)`	`720`
GraphFibLang	рекурсия, цикл, присваивание и печать	числа от 1 до 10	`1, 1, 2, 3, 5, 8, 13, 21, 34, 55`
GraphFibLang	ввод, цикл и накопление суммы	входное значение `5`	`15`
GraphStickLang	альтернативное представление чисел	палочные числа от 1 до 10	строки длины `1, 1, 2, 3, 5, 8, 13, 21, 34, 55`
MiniL	альтернативный синтаксис функций и блоков	`fact(6)`	`720`
MiniL	ввод, цикл и вызов функции	входное значение `5`	`15`

добавить другую модель исполнения, например регистровую машину или более сложное представление состояния программы. Тогда одна и та же спецификация языка сможет проверяться на разных способах исполнения.

Второе направление – трансляция не только в ПОЛИЗ, но и в другие промежуточные или целевые представления. Например, можно рассматривать одноадресный или многоадресный автокод, а также ассемблероподобное представление. В этом случае универсальный интерпретатор будет использоваться не только для запуска программ, но и как экспериментальная платформа для изучения трансляции.

Третье направление связано с развитием механизма семантических действий. Чем богаче набор действий, доступных внутри L-преобразователя, тем больше языковых средств можно выразить через графовую спецификацию. Сюда относятся более сложные типы данных, операции над составными объектами, расширенная работа с функциями и более гибкое управление областями видимости.

Наконец, отдельно нужно развивать диагностику. Пользователю полезно видеть не только сообщение об ошибке, но и место несоответствия входной программы заданным графам, текущую вершину L-графа, сформированный фрагмент ПОЛИЗ и историю выполненных действий. Такая визуальная отладка сделает систему не просто интерпретатором, а инструментом исследования языковых спецификаций [18].

VII. Заключение

В работе предложена и реализована архитектура универсального интерпретатора, в которой описание языка отделено от механизма исполнения. L-графы используются для представления лексических и синтаксических конструкций, а L-преобразователь с действиями задает перевод корректной программы в постфиксное внутреннее представление.

Разработанный прототип включает JSON-спецификацию языка, лексический и синтаксический анализ, компиляцию в стековый байткод, виртуальную машину и веб-интерфейс с графическим редактором. Работоспособность универсального интерпретатора проверена на минимальном языке одного числа и на более сложных примерах с функциями, рекурсией, циклами, вводом данных и альтернативным синтаксисом.

Полученные результаты показывают, что L-преобразователи могут служить основой для экспериментальной среды проектирования интерпретируемых языков. Наиболее полезным такой инструмент представляется в учебных и исследовательских задачах, а также при быстром прототипировании языковых конструкций. Текущая версия программного прототипа опубликована в репозитории Universal L-Interpreter на GitHub [19].

Список литературы

[1] Johnson S. C. Yacc: Yet Another Compiler-Compiler // Unix Programmer's Manual. – Murray Hill: Bell Laboratories, 1975.

[2] Parr T. The Definitive ANTLR 4 Reference. – Raleigh: Pragmatic Bookshelf, 2013.

[3] Lopez B., Ortin F., Noval J. Reflection as the Basis for Developing a Dynamic SoC Persistence System // Journal of Object Technology. – 2004. – Vol. 3, no. 8. – P. 121–145.

[4] Tusil J., Obdrzalek J. Minuska: Towards a Formally Verified Programming Language Framework. – 2024. – arXiv:2409.11530.

[5] Rosu G., Serbanuta T. F. An Overview of the K Semantic Framework // Journal of Logic and Algebraic Programming. – 2010. – Vol. 79, no. 6. – P. 397–434.

[6] Felleisen M., Findler R. B., Flatt M. Semantics Engineering with PLT Redex. – Cambridge: MIT Press, 2009.

[7] Kats L. C. L., Visser E. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs // Proceedings of OOPSLA 2010. – New York: ACM, 2010. – P. 444–463.

[8] Reynolds J. C. Definitional Interpreters for Higher-Order Programming Languages // Proceedings of the ACM Annual Conference. – New York: ACM, 1972. – P. 717–740.

[9] Wuerthinger T., Wimmer C., Woess A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., Wolczko M. One VM to Rule Them All // Proceedings of Onward! 2013. – New York: ACM, 2013. – P. 187–204.

[10] Жоголев Е. А. Интерпретатор ПОЛИЗ-63 // Журнал вычислительной математики и математической физики. – 1965. – Т. 5, № 1. – С. 67–76.

[11] Вылиток А. А. Описание формальных языков L-графами // International Journal of Open Information Technologies. – 2025. – Т. 13, № 11. – С. 1–5.

[12] Ли Цзямянь. Алгоритмы преобразования бесконтекстных грамматик в L-графы // International Journal of Open Information Technologies. – 2025. – Т. 13, № 11.

[13] Му Цзинъюань. Алгоритм преобразования бесконтекстных выражений в эквивалентные L-графы // International Journal of Open Information Technologies. – 2025. – Т. 13, № 11.

[14] Вылиток А. А., Зубова М. А., Мельников Б. Ф. Об одном расширении класса конечных автоматов для задания контекстно-свободных языков // Вестник Московского университета. Серия 15: Вычислительная математика и кибернетика. – 2013. – № 1. – С. 39–45.

[15] Вылиток А. А., Лай Вэньтао. Построение алгоритмов с помощью L-графов // Тезисы докладов научной конференции "Тихоновские чтения". – М.: МАКС Пресс, 2022. – С. 48.

[16] Вылиток А. А., Лай Вэньтао. Построение композиции L-преобразователей // Тезисы докладов научной конференции "Тихоновские чтения". – М.: МАКС Пресс, 2023. – С. 106.

[17] Лай Вэньтао. Конструирование алгоритмов и генерация программ на основе L-преобразователей: магистерская диссертация / науч. рук. А. А. Вылиток. – М.: МГУ имени М.В. Ломоносова, 2024.

[18] Николаев А. А. Разработка и реализация универсального интерпретатора на основе L-преобразователей: магистерская диссертация / науч. рук. А. А. Вылиток. – М.: МГУ имени М.В. Ломоносова, 2026.

[19] Universal L-Interpreter [Электронный ресурс] // GitHub. – URL: https://github.com/Nikas0604/Universal-L-interpreter.

On an Approach to the Implementation of a Universal Interpreter Based on L-Transducers

A.A. Nikolaev

***Abstract* – The paper presents the development of a universal interpreter that can be configured for a specific programming language using a dedicated graph-based specification. The lexical and syntactic levels of the language are represented by sets of L-graphs, while the rules for translating a program into an internal representation are defined by an L-transducer with actions. A postfix sequence of instructions is used as the intermediate representation and is executed by a stack-based virtual machine. The paper describes the general scheme of program processing, the language-specification format, and the architecture of the software prototype. The implementation includes a graphical editor, loading and validation of the language specification, lexical and syntactic analysis, generation of a program for the stack machine, and its execution. The proposed approach is illustrated by an example in which a program consists of a single number and produces that number incremented by one. The paper also discusses the supported constructs of input languages and directions for further development of the system.**

***Keywords* – formal language, L-graph, L-transducer, universal interpreter, stack machine, syntax analysis.**

References

[1] Johnson S. C. Yacc: Yet Another Compiler-Compiler // Unix Programmer's Manual. – Murray Hill: Bell Laboratories, 1975.

[2] Parr T. The Definitive ANTLR 4 Reference. – Raleigh: Pragmatic Bookshelf, 2013.

[3] Lopez B., Ortin F., Noval J. Reflection as the Basis for Developing a Dynamic SoC Persistence System // Journal of Object Technology. – 2004. – Vol. 3, no. 8. – P. 121–145.

[4] Tusil J., Obdrzalek J. Minuska: Towards a Formally Verified Programming Language Framework. – 2024. – arXiv:2409.11530.

[5] Rosu G., Serbanuta T. F. An Overview of the K Semantic Framework // Journal of Logic and Algebraic Programming. – 2010. – Vol. 79, no. 6. – P. 397–434.

[6] Felleisen M., Findler R. B., Flatt M. Semantics Engineering with PLT Redex. – Cambridge: MIT Press, 2009.

[7] Kats L. C. L., Visser E. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs // Proceedings of OOPSLA 2010. – New York: ACM, 2010. – P. 444–463.

[8] Reynolds J. C. Definitional Interpreters for Higher-Order Programming Languages // Proceedings of the ACM Annual Conference. – New York: ACM, 1972. – P. 717–740.

[9] Wuerthinger T., Wimmer C., Woess A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., Wolczko M. One VM to Rule Them All // Proceedings of Onward! 2013. – New York: ACM, 2013. – P. 187–204.

[10] Zhogolev Ye. A. The Interpreter POLIZ-63 // U.S.S.R. Computational Mathematics and Mathematical Physics. – 1965. – Vol. 5, no. 1. – P. 89–100.

[11] Vylitok A. A. Describing Formal Languages Using L-Graphs // International Journal of Open Information Technologies. – 2025. – Vol. 13, no. 11. – P. 1–5. (In Russian.)

[12] Li Jiamian. Algorithms for Transforming Context-Free Grammars into L-Graphs // International Journal of Open Information Technologies. – 2025. – Vol. 13, no. 11. – P. 6–13. (In Russian.)

[13] Mu Jingyuan. Algorithm for Transforming Context-Free Expressions into Equivalent L-Graphs // International Journal of Open Information Technologies. – 2025. – Vol. 13, no. 11. – P. 14–22. (In Russian.)

[14] Vylitok A. A., Zubova M. A., Melnikov B. F. On an Extension of the Class of Finite Automata for Defining Context-Free Languages // Vestnik of Moscow University. Series 15: Computational Mathematics and Cybernetics. – 2013. – No. 1. – P. 39–45. (In Russian.)

[15] Vylitok A. A., Lai Wentao. Constructing Algorithms Using L-Graphs // Abstracts of the Scientific Conference "Tikhonov Readings". – Moscow: MAKS Press, 2022. – P. 48. (In Russian.)

[16] Vylitok A. A., Lai Wentao. Constructing a Composition of L-Transducers // Abstracts of the Scientific Conference "Tikhonov Readings". – Moscow: MAKS Press, 2023. – P. 106. (In Russian.)

[17] Lai Wentao. Construction of Algorithms and Program Generation Based on L-Transducers: Master's Thesis / Supervisor A. A. Vylitok. – Moscow: Lomonosov Moscow State University, 2024. (In Russian.)

[18] Nikolaev A. A. Development and Implementation of a Universal Interpreter Based on L-Transducers: Master's Thesis / Supervisor A. A. Vylitok. – Moscow: Lomonosov Moscow State University, 2026. (In Russian.)

[19] Universal L-Interpreter [Electronic resource] // GitHub. – URL: https://github.com/Nikas0604/Universal-L-interpreter.